Which Feature Location Technique is Better?

Emily Hill[1], Alberto Bacchelli[2], Dave Binkley[3], Bogdan Dit[4], Dawn Lawrie[3], Rocco Oliveto[5]
[1]Montclair State University, Montclair, NJ, USA
[2]University of Lugano, Switzerland & Delft University of Technology, The Netherlands
[3]Loyola University Maryland, Baltimore, MD, USA
[4]The College of William and Mary, Williamsburg, VA, USA
[5]University of Molise, Pesche (IS), Italy
hillem@mail.montclair.edu, a.bacchelli@tudelft.nl, binkley@cs.loyola.edu,
bdit@cs.wm.edu, lawrie@cs.loyola.edu, rocco.oliveto@unimol.it

***Abstract*—Feature location is a fundamental step in software evolution tasks such as debugging, understanding, and reuse. Numerous automated and semi-automated feature location techniques (FLTs) have been proposed, but the question remains: How do we objectively determine which FLT is most effective? Existing evaluations frequently use bug fix data, which includes the location of the fix, but not what other code needs to be understood to make the fix. Existing evaluation measures such as precision, recall, effectiveness, mean average precision (MAP), and mean reciprocal rank (MRR) will not differentiate between a FLT that ranks higher these related elements over completely irrelevant ones. We propose an alternative measure of relevance based on the likelihood of a developer finding the bug fix locations from a ranked list of results. Our initial evaluation shows that by modeling user behavior, our proposed evaluation methodology can compare and evaluate FLTs fairly.**

***Index Terms*—Feature location, Concern location, Relevance measures, Empirical studies**

I. Introduction

Feature location in software, also called concept or concern location, is the process of identifying the source code elements, such as methods or files, that implement a user-observable feature [1], [2]. Feature location is a fundamental step in software evolution tasks such as debugging, understanding, and reuse. Researchers have proposed numerous automated and semi-automated feature location techniques (FLTs) [3], which usually recommend a ranked list of elements to be examined by a developer. Existing approaches to evaluating feature location techniques have predominantly used bug fixes automatically mined from source code repositories [3], [4]. The locations of the fix are likely relevant, but what about the code that must be understood to implement the fix?

Let us make the parallel that navigating a software system's source code is like traveling the globe. You are a developer in New York, trying to reach San Francisco, where you need to fix a bug. Recommender *A* gives you directions to St. Petersburg, Russia, which you immediately recognize as bad directions to an unrelated continent. Recommender *B* gives you directions to Los Angeles, which you follow, eventually realizing that you are still not that close to San Francisco. Finally, Recommender *C* gives you directions to Berkeley, and from there you can successfully navigate to your final destination. If Recommender *C* had been your first choice, you would have saved navigation time. This parallel illustrates the plight of a developer using FLTs in trying to locate source code during a software maintenance task in unfamiliar software. In this paper, we propose a way to differentiate between FLTs that send developers to Russia over Berkeley.

Going back to code, consider the following scenario with two FLTs, A and B. In the top 5 recommended code elements, FLT A recommends 4 methods that would need to be understood to implement the fix, and one fix location at rank 5. FLT B also recommends the fix location at rank 5, but the methods at ranks 1 through 4 are irrelevant (*i.e.*, are part of unrelated areas of the software system). Although A is intuitively better than B, the current measures commonly used to compare FLTs, *e.g.*, precision, recall, effectiveness, mean average precision (MAP), or mean reciprocal rank (MRR) [5], would mark them as equally effective.

An alternative to bug fix data is to have developers manually locate features in code [6], [7]. This process can be dependent on the annotators, and may change depending on the expertise, experience, and primary evolution task being undertaken. Prior attempts to create such gold sets have found very little agreement between annotators [7], which is also supported by anecdotal evidence from the authors' experiences in gathering gold sets. The main advantage of bug fix data is its objectivity: given a bug report, the bug fix confirms code locations that were actually changed to implement a modification to the feature as described by the bug report. Nevertheless, by using bug fix data we are faced with the challenge of fairly evaluating our hypothetical FLTs A and B. We support the use of bug fix data to objectively evaluate FLTs, but we propose an alternative measure of relevance based on the *likelihood* of a developer finding the fix locations.

In this paper, we introduce *rank topology*, a metric for comparing FLTs. We evaluate our proposed measure by running a FLT based on a state of the art Information Retrieval (IR) technique with different randomly generated lists of results that have the same ranks for the relevant code elements but contain other important code elements at different ranks. Our goal is to determine whether our rank-topology metric can distinguish between different lists where the relevant documents (*i.e.*, bug fixes) have exactly the same ranks.

II. The Rank Topology Approach

Our goal is to distinguish between FLTs that rank higher documents closely related to the bug fix than completely unrelated ones. Ideally we would like to mimic the behavior of a typical developer navigating the source code using the search results as a starting point [8], [9]. However, the accuracy of our estimates must be balanced with ease of use: if an evaluation metric is too complicated or difficult to use, it will not be widely adopted by the research community.

Thus, we propose a simple model of developer behavior. We assume the developer starts at the top of the ranked list of documents and investigates them in sequential order. At each rank, the developer has a choice of whether or not to explore the structural topology of the current document. As the developer explores the structural topology, she is constantly faced with the decision to continue forward or go back to the ranked list. In the following subsections, we formalize this decision making process and provide our current solution.

A. Proposed Rank Topology Framework

To help describe and compare FLTs, we introduce a two-part framework that provides a template for evaluating a FLT by dividing the search into two phases. In the first phase a developer considers, in rank order, the entries of the ranked list returned as the result of a search. When considering an entry, the developer may have some impression of the following k entries. For example, if when considering e_2, the next entry e_3, has a much lower score, then the developer may have the impression that it is worth considering e_2 longer than average, since the next entry appears irrelevant.

Thus, phase 2, which consists of the investigation of an entry e_i from the ranked list, begins with e_i and some impression of the subsequent entries (*e.g.*, e_{i+1}) in the ranked list. Phase 2 considers e_i and the code artifacts connected to it. The connections are captured as a set of relations (*e.g.*, callees, callers, textual similarity, within the same file, (data) depends on, *etc.*). Each relation can have a weight associated with it. For example, callers might be valued more than callees. We make the simplifying assumption that developers do not revisit a code artifact; thus, they explore a subtree of a spanning tree of the graph defined by the structural relations.

At each code artifact, one of the following two decisions is made: bail or next. If the developer opts to bail on the current structural search, they return immediately to the next entry in the ranked list. Alternatively, the next code artifact can be chosen from the set of structurally connected code artifacts to the structural path being traversed. We assume the developer only chooses to investigate code artifacts that appear "interesting" (*i.e.*, that have a high enough value). In either case the impression (of the value of bailing) is updated. This value can be increased if nothing useful is being encountered or it can be decremented if phase 2 is going well.

B. Modeling an Imperfect User

One of the challenges in using a rank topology measure is in determining how *smart* the developer is. An *omniscient* developer would make no wrong choices and explore every profitable structural edge no matter how tenuously related to the bug description. For example, consider JabRef[1], an open source Java bibliography reference manager, and its bug #1297576 with query *"Printing of entry preview"*. All 20 of the fix locations are just 4 or 5 structural "hops" (*i.e.*, edges) away from the very first result, the `actionPerformed` method in the `PrintAction` class. While the `PrintAction` class seems relevant to the bug, and it is understandable that some of the relevant documents are located nearby in the program structure, some of the links are not obvious. For example, navigating these structural paths requires navigating through a generic `openWindow` method that launches a number of actions within JabRef, and then to the generic `get(String)` method in the `JabRefPreferences` class. The presence of these generic, highly-connected methods makes it trivial for an omniscient user to navigate to most bug fix locations within 4-6 structural hops of the top ranked document. Although most developers are not omniscient, the perfect omniscient user provides an upper bound on the limits of structural topology in finding fix locations.

Developers who are not omniscient will not make generic structural hops unless there is strong evidence to do so. Prior work applying information foraging theory to developer behavior has shown that simple textual information, such as the cosine similarity between a method and a query, closely models actual developer behavior when debugging [8]. Thus, we propose a semi-intelligent user that only follows a structural link if the next method exhibits textual clues.

C. The Proposed Rank Topology Metric

For each bug fix location, we determine the shortest number of hops required to find it in terms of structural topology and the ranked list. The result set is the minimum cost of navigating to each fix location. For simplicity, in the current work the cost of each rank jump and each structural edge is one. If a developer navigates the structural topology from a method to another method in the same class, we assign a cost of two (one to navigate to the class and one to get to the desired method). Textual information can be used to approximate whether a developer would recognize a structural link and follow it. If the textual relevance to the query is below a certain threshold (*i.e.*, if the developer is unlikely to explore it), the structural edge is ignored.

In this initial work, we make the simplifying assumption of exploring every ranked document, and not considering all the wrong explorations before arriving at a fix location. For example, a fix location may be 5 structural hops away from the third ranked document, giving a total cost of 8. This distance does not consider all the elements explored with rank 1 or 2 before arriving at rank 3 when calculating the cost; it only considers the number of elements to reach the current rank in the list (3, in this case) and the direct structural cost of the path to the fix. The inverse of this cost forms our metric.

[1] http://jabref.sourceforge.net/

Our proposed *rank topology metric* is inspired by average precision (AP), which is commonly used to calculate MAP scores in evaluating IR systems [5]. Like AP, our metric involves the average of the inverse ranks for each relevant document. However, unlike the AP score, the ranks used are based on the costs using structural topology and rank.

D. Current Implementation

To implement the structural topology, we experimented with using method call and type hierarchy information as undirected graphs. Call graphs were extracted using Eclipse's statically available call hierarchy functionality. Next, we used JGraphT's implementation (http://jgrapht.org/) of the Floyd-Warshall algorithm to find all shortest paths from the bug fix locations to every other method in the program. We use a very simple form of textual information, cosine similarity, to determine textual relevance of each method to the query. The cosine similarity is computed using the Vector Space Model (VSM) where term weights are calculated using term frequency-inverse document frequency (tf-idf). All of these components to the rank-topology metric are available by third party libraries, and need not be implemented from scratch.

III. Preliminary Evaluation

Our premise is that existing IR measures such as precision, recall, and MAP cannot differentiate between techniques that rank related versus irrelevant documents above the fix locations. To evaluate if our proposed rank-topology measure captures this distinction, we compare a state of the art IR technique with a randomly ordered list of methods in the program with the same relevant fix locations at the exact same ranks as the IR technique. Thus, we control that the only source of variation is in the ordering of the non-relevant methods (*i.e.*, the methods not involved in the bug fix). By crafting the randomly ordered results in this way, we guarantee the results are indistinguishable by current evaluation measures.

We selected a small set of bugs from the JabRef program in the SEMERU dataset [3], [10]. To determine if our rank topology metric depends on how relevant results are initially ranked, we selected bugs from three different types of ranked lists: one where relevant documents appeared in the top five, one where relevant documents appeared from five to ten, and one where relevant documents first appeared around rank 100. We selected two queries (*i.e.*, bugs) of each type. None of the selected bugs ranked a relevant document in the first position.

The retrieval method used to produce these ranked lists was the Query Likelihood Model (QLM). The model was selected because it does not use VSM, which underlies our rank-topology technique of modeling the imperfect user. Instead, QLM relies on language models to estimate the probability that a document generates a query. Documents with higher probabilities of generating the query are ranked higher. Prior work showed that this model performs well in FLTs [4].

A. Effect of Structural Topology on an Omniscient User

One of the parameters to the rank topology approach is what structural information to explore to find fixes in close proximity to the ranked entries. When navigating source code, developers have access to caller and callee information in the call hierarchy, can jump to other classes in the type hierarchy, or can jump through its class to other methods within the same class file. Before investigating whether our rank topology metric can distinguish between random and QLM, we first explore the effect that different structural topology can have on the results.

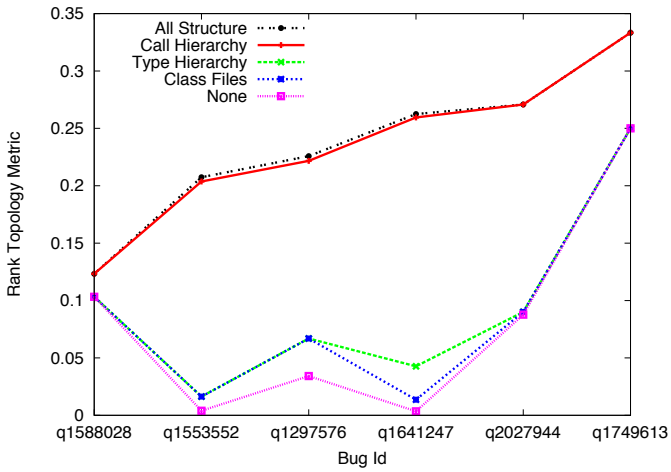

Figure 1. Effect of program structure on the rank topology metric for each JabRef bug used in the case study. Caller information has the biggest impact on finding bug fixes structurally close to the retrieved results.

Figure 1 shows the rank topology (RT) measure for the six JabRef bugs in our case study, sorted by increasing RT scores when using all structural information. The pink line at the bottom uses no structural information, and represents the RT scores for the original raw document ranks. Caller and callee information is combined into the call hierarchy. The class files structure adds an edge to the topology from every class to every method, allowing a method to jump to any other method in the same class using 2 hops. Finally, type hierarchy information adds to the edges introduced by class files with additional edges between super and subclasses.

As highlighted by Figure 1, call information is the dominant connector among ranked documents and bug fix locations. Adding class and type hierarchy information results in a very small improvement overall. Although in the remaining experiments we use all the structure information, the results would be similar if we used only call information.

B. Modeling the Imperfect User

The purpose of our case study is to determine if a rank-topology measure can distinguish between QLM and a randomly ordered list of results with exactly the same relevant documents at the same ranks. Figure 2 shows the rank topology metric for each bug in the study. The top line in the figure represents an omniscient user that finds the shortest possible path through the structural topology, irrespective of textual clues. The solid red line below this represents the state of the art IR technique, QLM, with a textual threshold based on

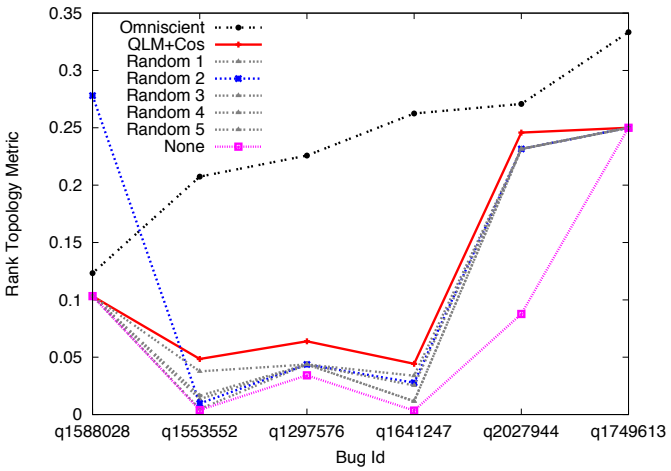

Figure 2. For each JabRef bug used in the case study, the proposed rank topology metric can differentiate between a state of the art IR technique (QLM+Cos) and 4 randomly ordered rank lists (1–5). QLM and random would be indistinguishable using existing IR measures, and would appear as the "None" line, which only includes ranks with no topology.

the VSM cosine similarity. A structural edge is only added to the topology if the similarity of both methods is greater than the median of all the scores returned for that query (*i.e.*, bug). We selected the median, rather than a raw threshold such as .5, because the actual range of cosine values is highly dependent on the nature of the query. We assume that if a method's relevance score is over the threshold, the edges to related classes (in the Files and Type Hierarchy structures) also exist.

The next 5 dotted lines in gray and blue are randomly generated lists, which also use the median cosine score to prune the structural topology. The pink bottom line uses the raw ranks alone with no structural topology, labelled "None". As expected, Figure 2 illustrates that our proposed rank topology metric differentiates between a randomly generated list of results and QLM. One of the randomly generated lists, Random 2 (in blue), significantly outperforms even an omniscient user exploring QLM's rankings for one bug. These results demonstrate that traditional IR measures such as precision, recall, MRR, and MAP can give a misleading picture of the effectiveness of a feature location technique, when a technique can produce a ranked list that allows the developer to navigate to a fix location quicker. For future work, we would like to study actual user navigation habits, to see if our proposed rank topology metric accurately reflects differences between ranked lists for software maintenance.

IV. Related Work

Existing FLTs can be classified by the underlying information they use, such as static, dynamic, textual, or historical information [3]. FLTs based on static analysis examine structural information such as control or data flow dependencies, and may incorporate textual information in comments and identifiers in the code. One such technique uses information foraging theory to analyze how developers navigate source code when fixing bugs [8]. Although our rank topology metric is inspired by the same theory, our aim is to quantify the effort required by developers when analyzing the suggestions of a FLT by defining a new measure of relevance that can better analyze and compare the performance of FLTs.

Another closely related work by Petrenko and Rajlich [11] is similar to the evaluation metric proposed in this paper. Specifically, the authors propose using an IR technique to identify bug fix locations. If the IR technique fails to identify a fix location, the developer can decide either to reformulate the query, or to identify a "focus method" (*i.e.*, a method related to the fix). In the latter case, the developer navigates program dependencies starting from the "focus method" to identify fix locations. In our current work, we use a similar infrastructure but from a different prospective. Whereas Petrenko and Rajlich [11] use dependencies to speed up the process of feature location and avoid query reformulation, we analyze dependencies to better compare and assess FLTs in a semi-automatic usage scenario.

V. Conclusion and Future Work

In this paper we presented a *rank topology metric* to fairly compare feature location techniques (FLTs). In a small case study, we demonstrated that our rank topology metric can differentiate between a state-of-the-art IR technique and a randomly ordered list of results with the same relevant results at the exact same ranks. These two techniques would be indistinguishable using existing IR measures commonly used to evaluate FLTs. For future work we will study how closely the proposed rank topology measure mimics an actual developer in practice when using FLTs during corrective maintenance. We also plan to investigate the feasibility of more closely modeling user navigation behavior from source code search results.

References

[1] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *ICSE*, 1993.

[2] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *IWPC*, 2002.

[3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *J. Soft. Maint. Evo.: Research & Practice*, vol. 25, no. 1, pp. 53–95, 2013.

[4] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *MSR*, 2011.

[5] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY: Cambridge University Press, 2008.

[6] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *AOSD*, 2007.

[7] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock, "An empirical study of the concept assignment problem," School of Computer Science, McGill University, Tech. Rep. SOCS-TR-2007.3, Jun. 2007, http://www.cs.mcgill.ca/~martin/concerns/.

[8] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE TSE*, vol. 39, no. 2, 2013.

[9] A. von Mayrhauser and A. M. Vans, "Program understanding behavior during debugging of large scale software," in *ESP Workshop*, 1997.

[10] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. H. Kagdi, "A dataset from change history to support evaluation of software maintenance tasks," in *MSR*, 2013.

[11] M. Petrenko and V. Rajlich, "Concept location using program dependencies and information retrieval (DepIR)," *Inf. Softw. Tech.*, vol. 55, no. 4, 2013.